Proposal of Transaction on RAMCloud

rev0.62

06 Oct. 2013

Satoshi Matsushita

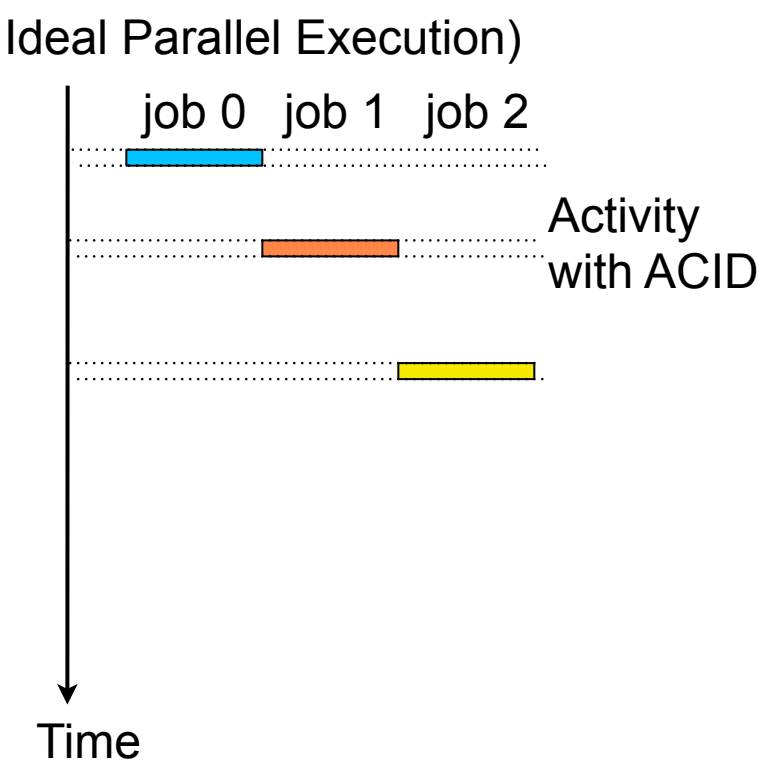
Problem Statement

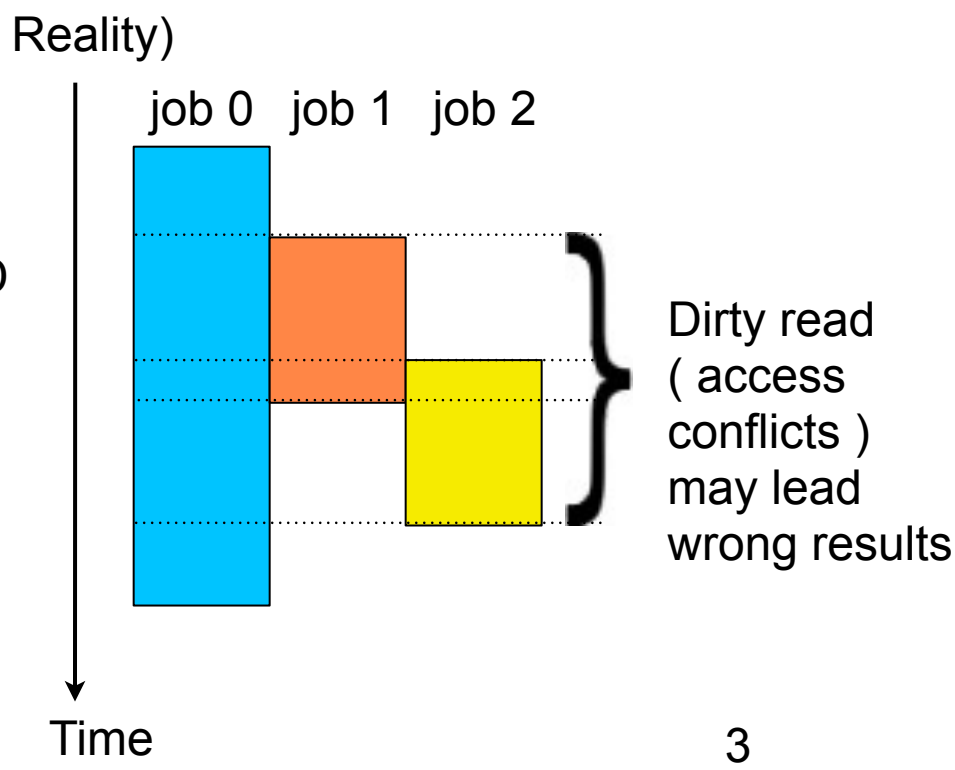
- Introduce transaction to RAMCloud
- What is ‘Transaction’?
  - Resolve problems in parallel execution.
- What is the characteristics of transaction?

To Be Added

Problem in Parallel Execution

- Resource access conflict occurs in parallel execution
- Requirement to avoid the problem
  - ACID: (*atomicity*, *consistency*, *isolation*, *durability*)
  - CAP Theorem: (Can relax partition tolerance) - discuss later



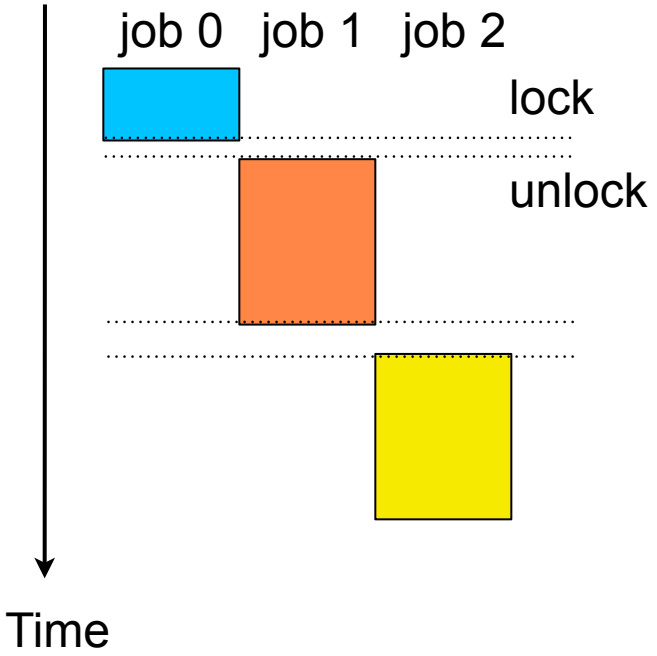


3

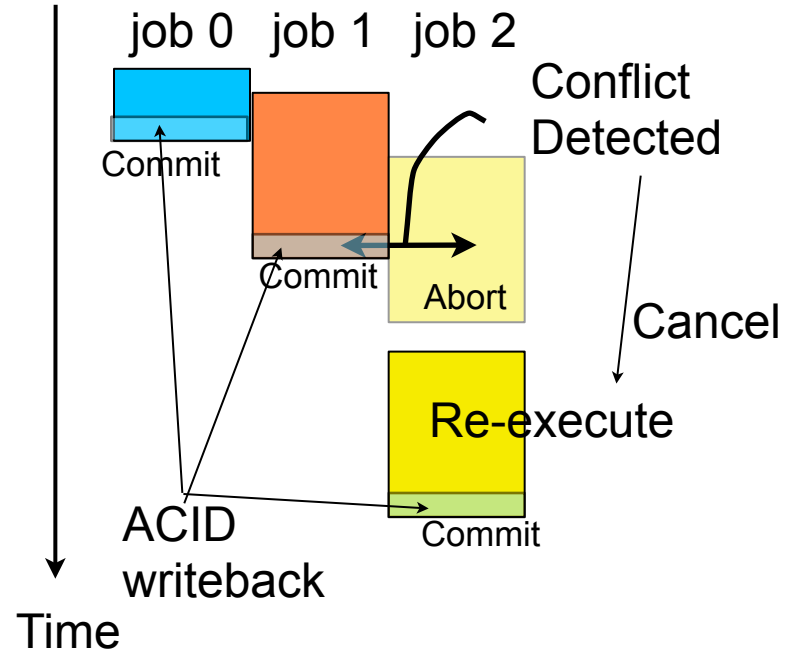
S. Matsushita, 10/17/2013, rev. 0.62

Conflict Solutions

Pessimistic Lock)



Optimistic Lock)



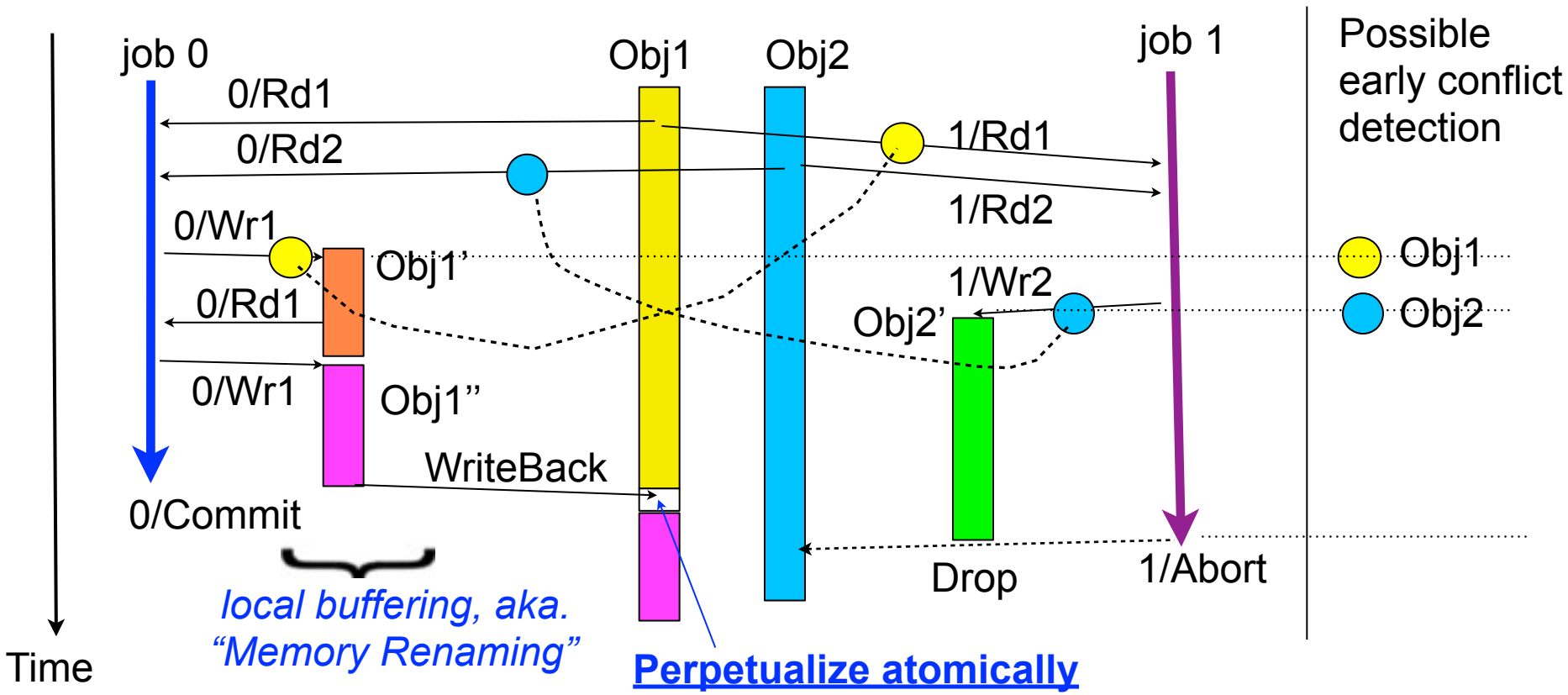
Text

Pros & Cons)
- Lower parallelism with giant locks
- Dead lock prone with fine locks
- Need releasing lock with node crash

Pros & Cons)
- Need conflict detection logic
- Lower Performance loss by frequent conflicts
- Alternatives in abort detection

Optimistic Lock: General Solution

- Conflict detection of true dependencies: RAW (Read after Write)
- Renaming false dependencies : WAR, WAR
  - Common technique in parallel execution such as Speculative MT, Transactional Mem., RDBM



S. Matsushita, 10/17/2013, rev. 0.62

Assumptions and Strategies

Application Specific)

- Transaction life varies between short to long
    - Try early conflict detection avoiding livelock
- Small probability of conflicts
    - Use optimistic lock based design
    - Otherwise use pessimistic lock at user level
- Small probability of node failure during a transaction
    - Involve small number of different nodes in a transaction

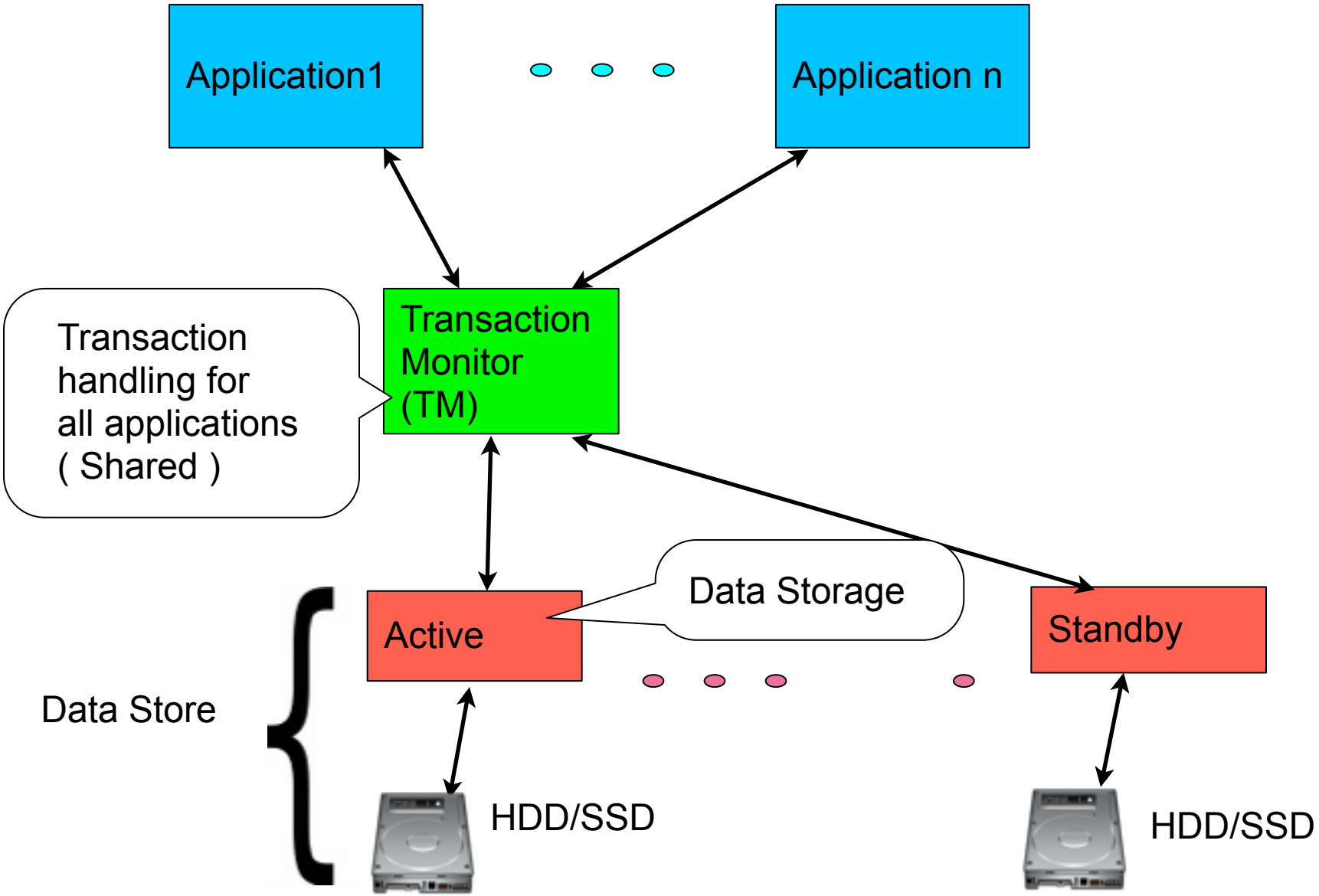
RAMCloud Specific)

- Faster crash recovery around 1 sec
    - Can yield to blocking algorithm to prevent corner cases

Note)

- CAP Theorem
  - Means: Consistency, Availability, Partition-tolerance
  - RAMCloud natively does not have partition tolerance, only the partition where coordinator exists works.
- Multiphase Commit
  - If we can allow waiting for node recovery, two phase commit works.
  - Since the blockage is not realistic, couple of non-blocking commit algorithm have been introduced:
    - Consensus (Paxos, Raft): Always live majority hides node crash
    - Multiphase Commit - prevent commit blockage
      - Quorum Commit: Majority side works during partitioning
      - Three phase commit - still it is not easy to detect failure mode.
      - Paxos commit, etc

Traditional Transaction System



Traditional Transaction: Sharding

- Problems)
  - Not easy to design field in record
  - Not always possible to allocate independent shard

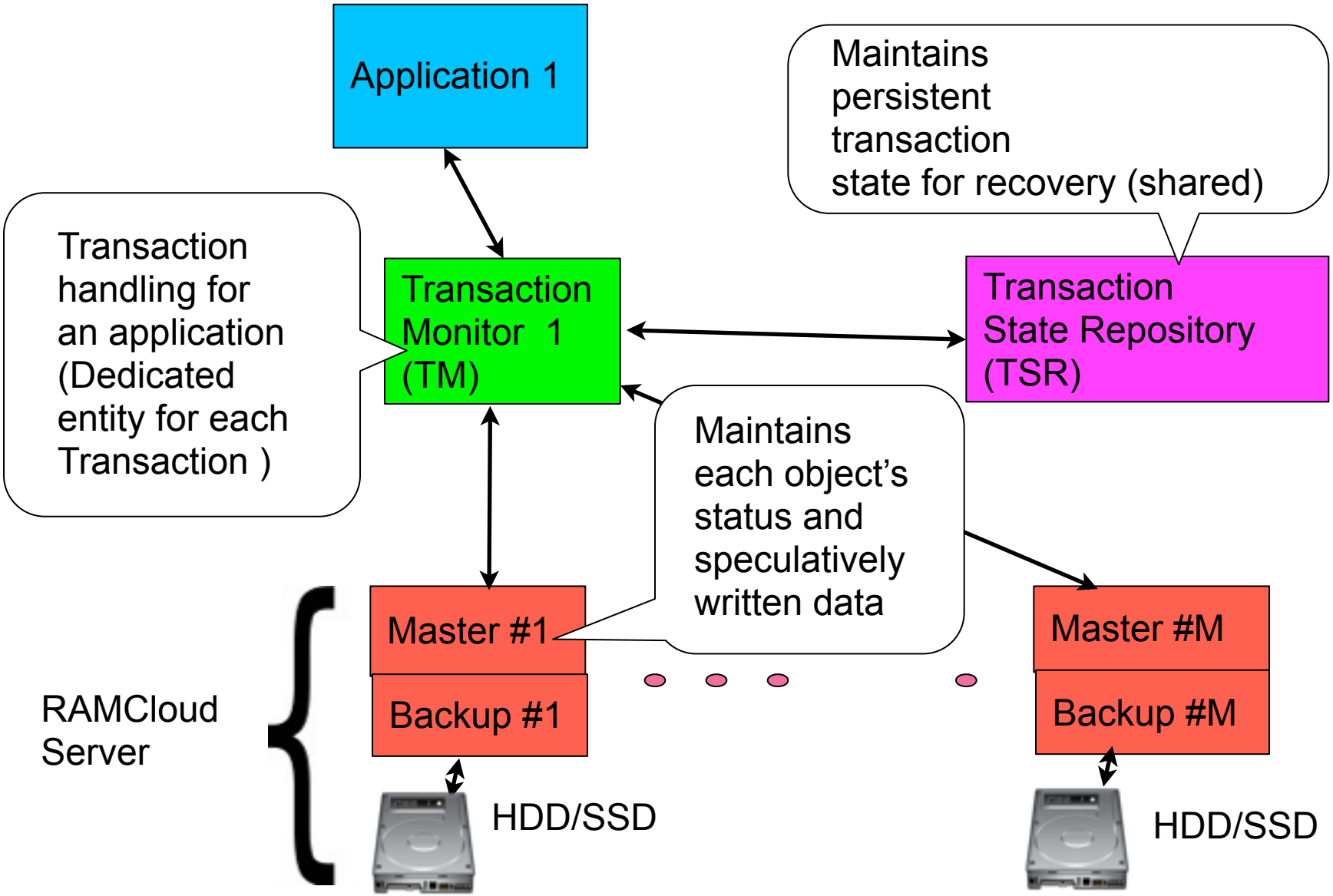
To Be Added

Traditional Transaction: Sinfonia

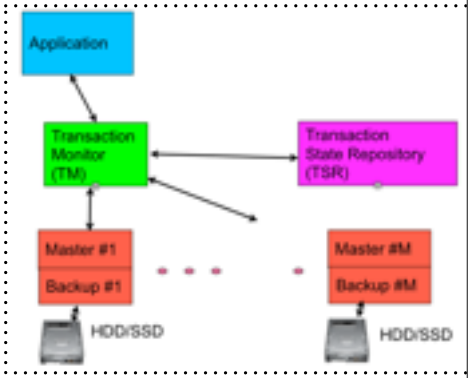
- Conditional commit (two phase commit) only
  - Delay inquiry to all relevant nodes at commit time

To Be Added

Proposal: Components



Components - Functions

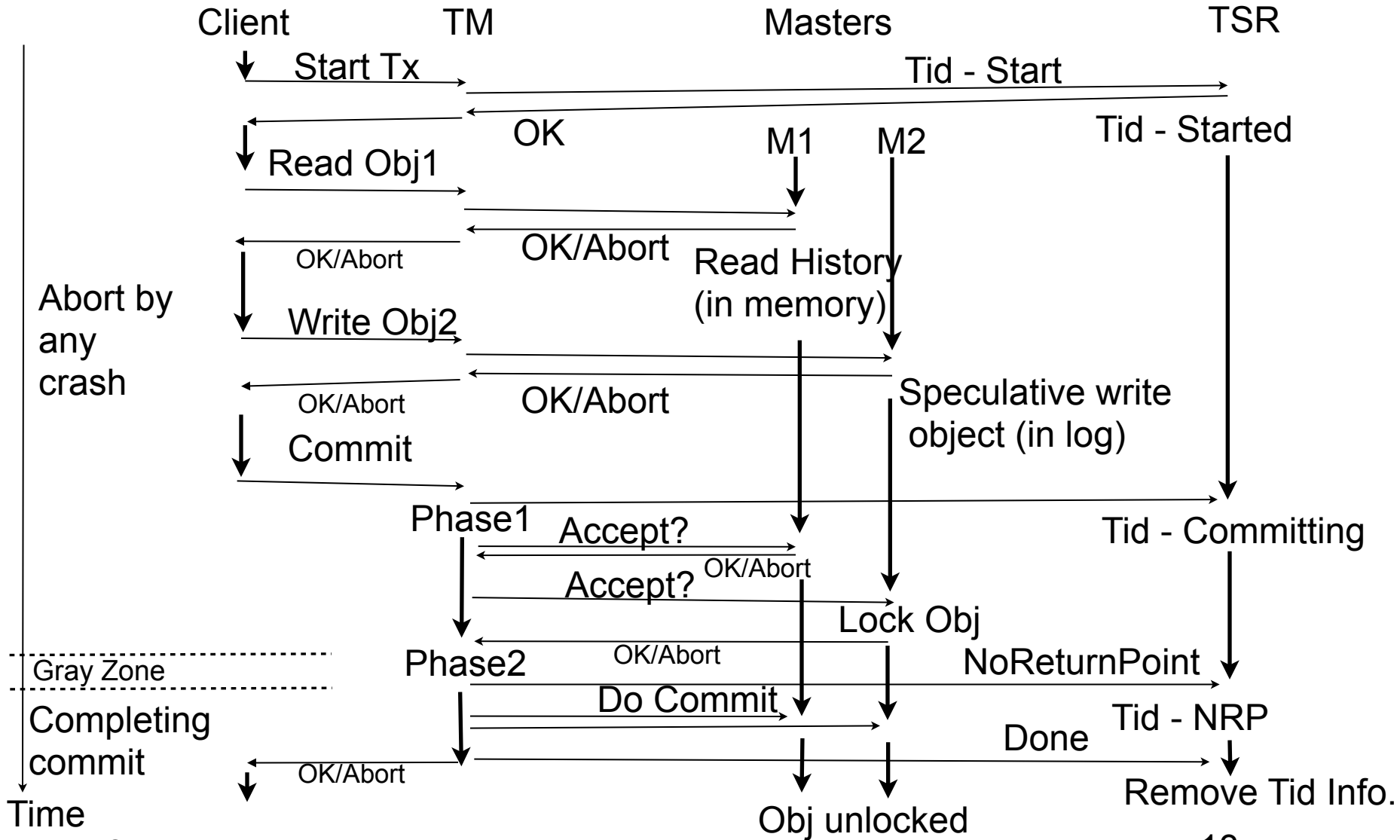


- If client application is restarted immediately (by coordinator, etc), TM can be implemented in client library.

Functions	TM:Trans. Monitor	TSR:Trans. State Repo.	Master	Coordinator
Normal Op.	Generate unique Transaction ID. Keep track objects states. 2phase commit coordination.	Store global status of a transaction persistently	Keep object s' status and temporal data, return appropriate data	Maintain crash information and TM identifier.
At Recovery	Continue 2phase commit (**resource unlock**)	TM accesses the transaction status	Respond TM to complete commit/abort	Restart TM, or notice TM crashed node.
Possible location	Client library, Client node, or Master	Master node as a normal table.	Master node	Coordinator

Basic Flow: Life of a Transaction

- Define Transaction priority uniquely with Tid: Transaction ID



Detailed discussion: outline

1. Client API
2. Conflict Management
   i. Resolution at object access with transaction priority
   ii. TMid/Tid for unique global transaction order
   iii. Timeout to avoid deadlock
3. Commit - transition from non-blocking to blocking
   (Gray zone solution)
4. Recovery
   i. Cleaning up by abort or completing commit
   ii. TM implementation
   service process or library - depends on client recovery
   iii. TSR implementation - in a normal table
5. Implementation Control / Data structure
6. Optimization
   i. Callback instead of piggyback
   ii. Separate key/state and data for objects in log

1. Client API

- Start Transaction
  - tx_start(&tid); // return new tid
- Object Access
  - tx_read(tid, tableId, key, &buf, &state...);
  - tx_write (tid, tableId, key, &buf, &state...);
  - tx_remove(), tx_multi-...(),
    We can make tx_read, tx_write by default using tid=0
    for non transactional operation.
- Commit Transaction
  - tx_commit(tid, &state);
  - tx_abort(tid, &state);
- Status
  - tx_status(tid, &state); // return current transaction state

2. Truth Table of Conflicts Management

- Older transaction id wins at data access
- Provides only shared reads: can detect Read/Read conflict with dummy write: Rd (Obj1) with Wr(Dummy1)

Tid 1 (Older) < Tid 2 (Younger)

operation mode	Tid 1	Tid 2	winner
mode1	read	read	both
mode2	Not Supported	read	Tid 1
both modes	read	write	Tid 1
both modes	write	read	Tid 1
both modes	write	write	Tid 1

S. Matsushita, 10/17/2013, rev. 0.62

Tid, TMid

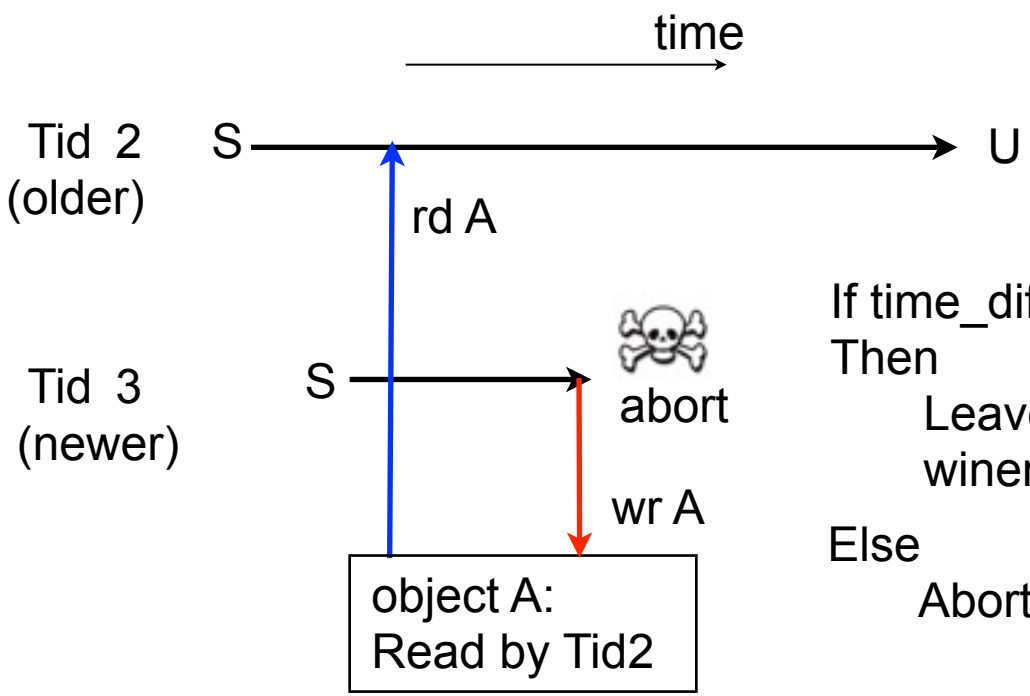
- TMid is given by coordinator at TM startup
- Tid
  - Define Tid = [TMid, TM-localtime] at a transaction generation // note: [a, b] = concatenation of ‘a’ and ‘b’
  - Compare TMid only when local time is the same
  - Preciseness is not needed, because Tid is just a priority to decide winner transaction at object access time.

S. Matsushita, 10/17/2013, rev. 0.62

Conflict management at object access

- Compares Tid in Master. Abort newer Tid immediately.
  (Traditional technique in DBMS)
- Timeout to avoid deadlock by incorrect code or client crash, which freezes the oldest transaction.

Notation)
S: Started
A: Aborted
U: Uncommitted
(Speculatively running)

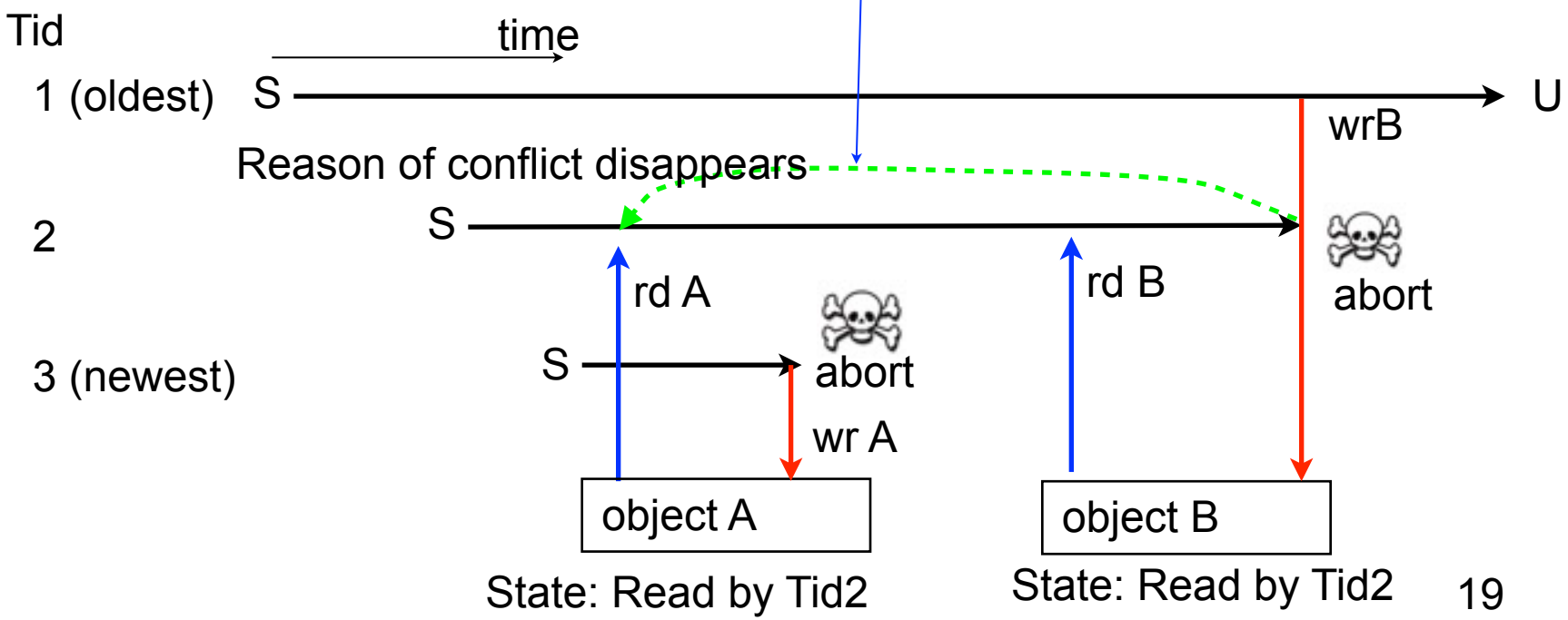


If time_difference(Tid3, Tid2) > Tout
Then
Leaves both alive and decides winer at commit time.
Else
Abort transaction with newer Tid

S. Matsushita, 10/17/2013, rev. 0.62

Issues - False abort/Status piggyback

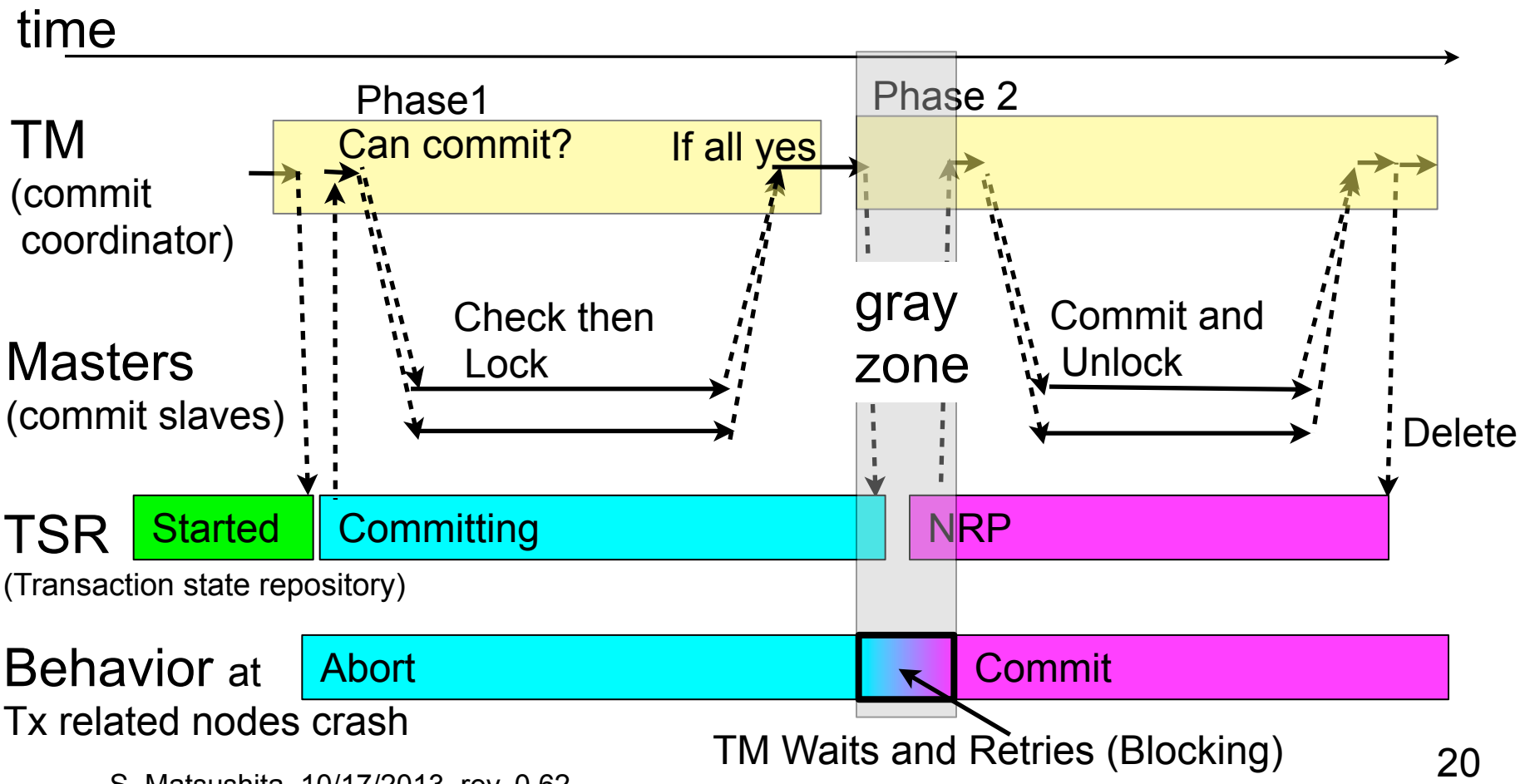
- False Abort: the conflict which aborted Tid3 disappears when Tid2 is aborted later.
  - Chain reaction of false abort may occur
  - Leave it because provability of false abort is small.
- Abort notified as status return (piggyback).
  - Tid2 is not aborted by Tid1-write, but by some request in the future (Needs callback to optimize)



S. Matsushita, 10/17/2013, rev. 0.62

3. Commit - Two phase commit

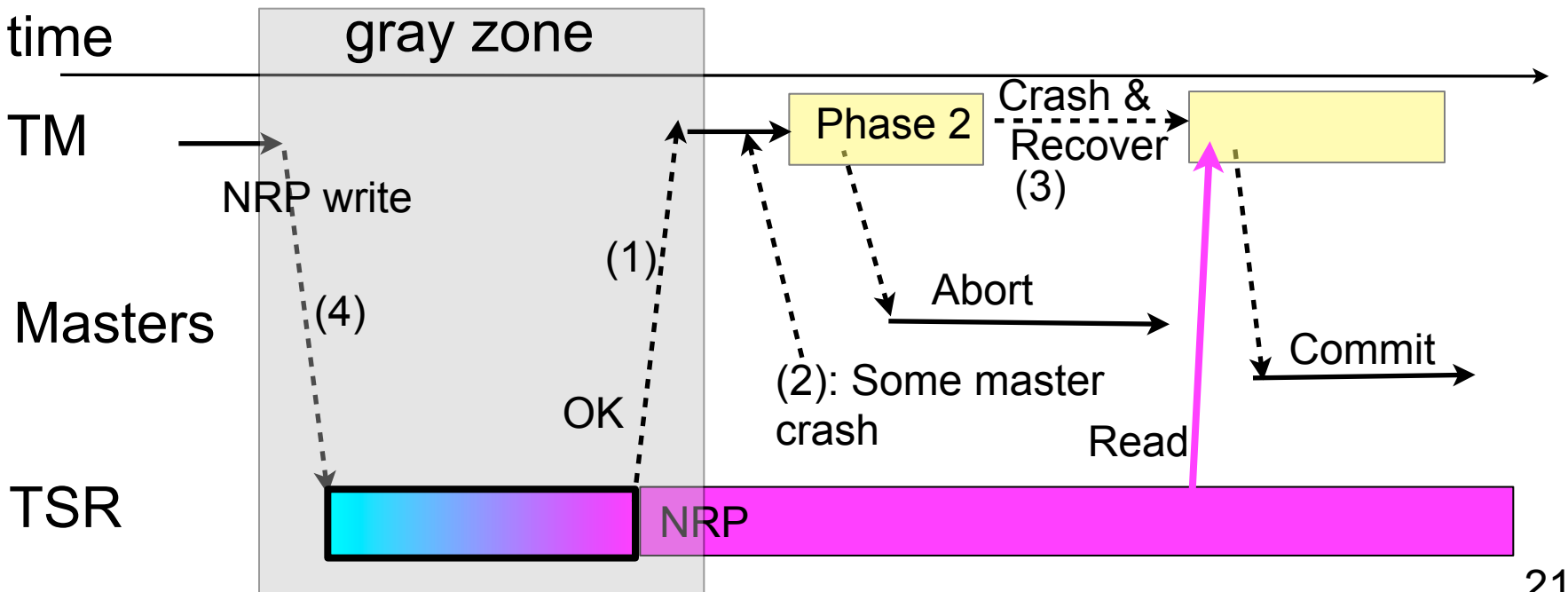
- TM coordinates commit operation
- Save durable state in TSR
  - Committing: unlock object by abort (optimization)
  - NRP: no-return-point for durable transition to commit



S. Matsushita, 10/17/2013, rev. 0.62

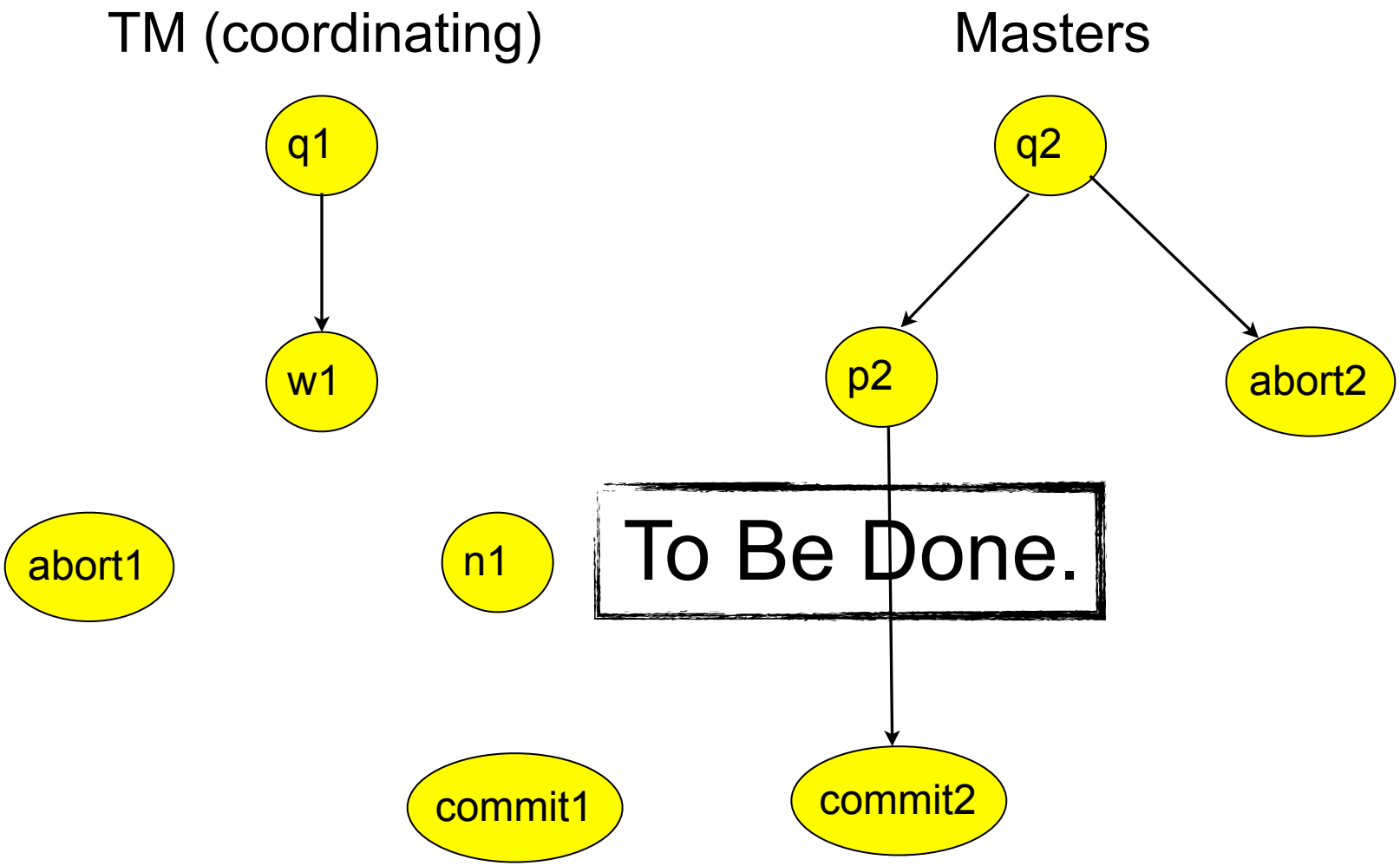
3. Commit - Racing conditions

- Racing condition:
  - After NRP is written, TM start aborting in Phase2 due to (1) lost 'OK' or (2) relevant node crash
    - Then TM crashes and recovered TM read NRP and start commit.
    - (1) cannot be distinguished from (4) lost NRP req
- Solution
  - NRP is idempotent: TM retries (4) and waits (1)
  - If TM failed retry, TM reads TSR after enough timeout to decides behavior.
  - After initiating (4), TM stop aborting Tx by relevant node crash.



S. Matsushita, 10/17/2013, rev. 0.62

3. Commit - State Machine



Ref: A Formal Model of Crash Recovery in a Distributed System, Dale Skeen and Michael Stonebraker, 1983

S. Matsushita, 10/17/2013, rev. 0.62

4. Crash Recovery - Clean up

- TM crash
  - completes commit/abort
    - Commits transaction if NRP is found. Otherwise abort transaction.
    - Fast restart required because other clients are blocked by accessing the locked objects
- Server crash
  - Reconstruct hash and object status in memory from log
- TSR crash
  - Recover status of transactions

5. Implementation Alternatives

- TM - item 1 seems simplest and good for performance.
  1. In client library such as crt0.
     - Pros) Application (Tire2, Tire3) needs to be recovered to continue web service.
     - Cons) Need client recovery mechanism by coordinator
  2. In a master
     - Need a location decision and lookup by coordinator
     - Cons) Extra access latency and network traffic by additional hop in data access.
  3. In a separate process/thread in a client node
     - Need another recovery mechanism
     - Cons) Extra latency by process communication and dispatch
- TSR
  - In a master with defining a table and save transaction state as a normal object.

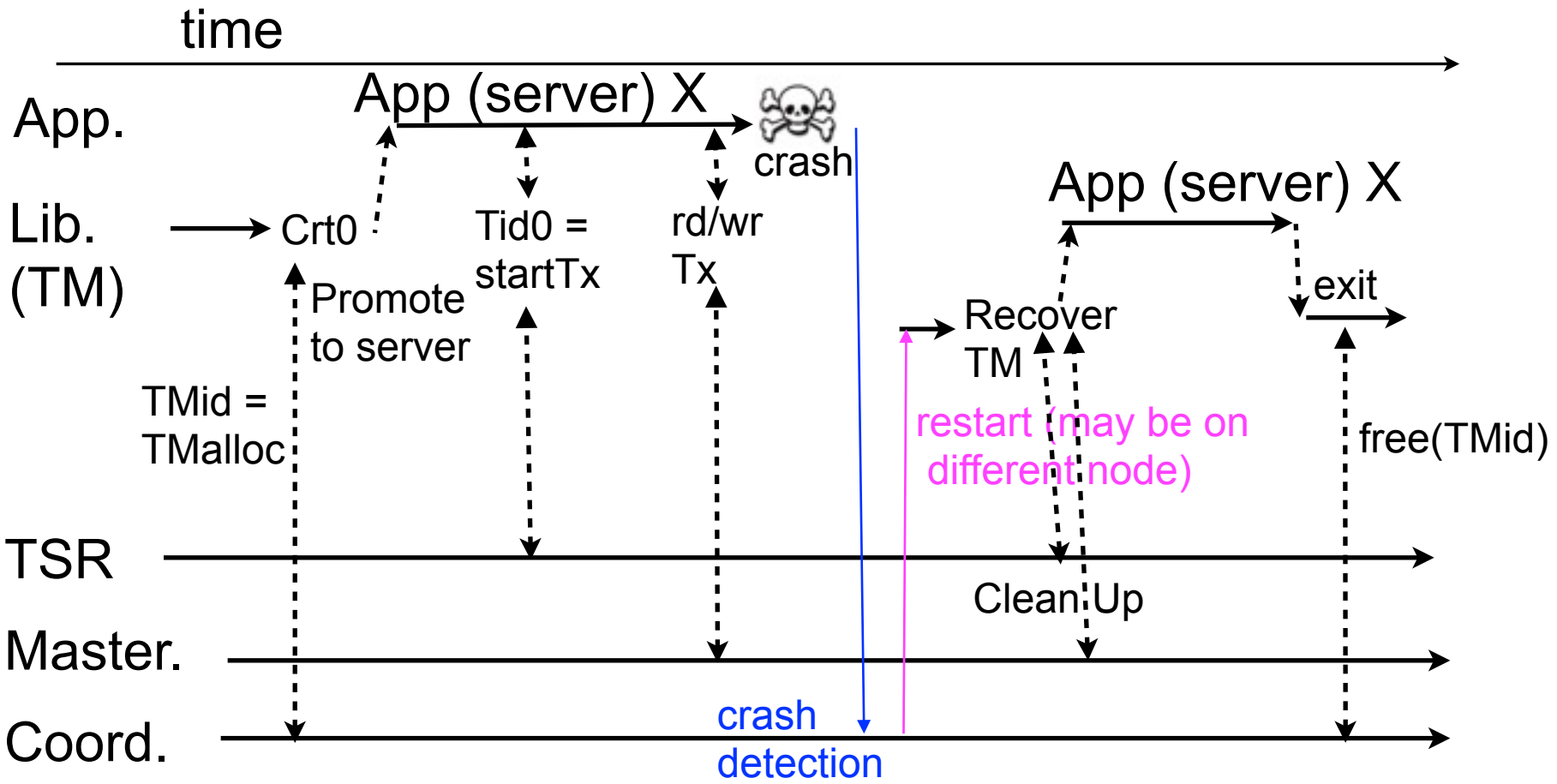
5. Implementation Proposal

- TM as client library
  - Coordinator detects client failure and restarts
  - Naming issue: ‘It would better to call an application promote to server by requesting recovery to coordinator’.
- TMid given by coordinator
  - Generate: Tid = [TMid, TM’s local time]
  - Timed loosely correct TM’s local time

- TSR as a specific table
  - (Key, Value) = (Tid, TransactionState)
  - How to find active transactions associated to a TMid?
    - Range query : [TMid, time min] to [TMid, time max]
    - Other object : (TMid, list_of_Tids)

5. Implement TM in Client Library

- Crt0 contacts coordinator to get TMid and register application info. for recovery.
- User can modify transaction algorithm by modifying library.



S. Matsushita, 10/17/2013, rev. 0.62

5. TM Data Structure

To Be Done.

Master Data Structure

To Be Done.